

# CUDA Library for T<sup>2</sup>OURNAMINT

Yajaira Gonzalez Gonzalez, Jeffrey C. McHarg  
MIT Lincoln Laboratory  
Lexington, Massachusetts, USA  
{yajaira.gonzalez, mcharg@ll.mit.edu}

**Abstract** — We present the design of an optimized CUDA Library used in T<sup>2</sup>OURNAMINT, an Electronic Warfare testing system designed to support many-on-many, hardware-in-the-loop test capabilities. The CUDA library was designed to handle the processing of environmental effects such as secondary target injection, clutter modeling, and receiver thermal noise.

**Keywords**—GPU, CUDA, Accelerator, Library, Radar Signal Processing, Tesla K40, Kepler, T<sup>2</sup>OURNAMINT.

## I. INTRODUCTION

T<sup>2</sup>OURNAMINT, which stands for *Testing Theater Operations Using Real-time Networks Achieving Multiple Interconnected Nodal Tactics*, is a real-time environment simulator for many-on-many, hardware-in-the-loop electronic warfare (EW) testing. T<sup>2</sup>OURNAMINT has been designed and developed at MIT Lincoln Laboratory and is envisioned to be the central link between multiple hardware-in-the-loop emulators and test chambers at the NAVAIR facility.

A realistic EW testing system needs to be able to manage, model, and represent a large number of environmental and system interactions. Those interactions include primary and secondary interactions. Primary interactions are those occurring between key systems under test, such as the emulation of radars tracking the trajectory of potential threats and the response of those threats to the radar signals. These interactions are expected to be small in number (tens of hardware systems) and require high fidelity hardware processing. Secondary interactions are what we will call environmental interactions that affect and impact the systems under test. These interactions include, but are not limited to, the simulation of additional non-threatening targets, clutter effects, and receiver limitations. In general, secondary types of interactions are those that are large in number (hundreds of objects and/or distributed responses in range) and require fast processing that can be performed at the software level. This paper focuses on how we manage the secondary types of interactions within T<sup>2</sup>OURNAMINT. To achieve high performance and realize acceptable processing speeds we leverage the use of an accelerator, the NVIDIA Tesla K40 graphics processing unit (GPU), programmed via the use of NVIDIA's Compute Unified Architecture (CUDA) API. In efforts to keep the accelerator support transferable to other projects and effortless in its usage, the functionality was built as a library.

## II. GPU SUPPORT FOR T<sup>2</sup>OURNAMINT

The GPU's role in T<sup>2</sup>OURNAMINT is to transform an incoming signal in a way that reflects specified environmental

effects for a given test sensor and test scenario. Environmental effects supported are secondary targets, clutter distributions, and receiver noise modeling. After such effects are computed based on the original signal, the resulting signal is sent back to the T<sup>2</sup>OURNAMINT software for further routing.

As shown in Figure 1, the GPU's role is to generate an IQ uncompressed response that is representative of the specified environment. To do this, the controlling software sends target descriptors to the GPU specifying target positions and gains. The GPU then converts these descriptors into target range profiles. If clutter is to be injected, relevant clutter parameters that describe the type of clutter, clutter distribution within a patch and wind speeds are also sent to the GPU for the generation of a clutter model. (The clutter model currently supported on T<sup>2</sup>OURNAMINT is a simplified version of the Land Clutter model proposed by Billingsley [1]). Once clutter effects and target effects have been combined, they are convolved with the transmit waveform on a pulse-by-pulse basis to distribute the energy of the original signal in time based on the given environment. In order to match processing gains within the T<sup>2</sup>OURNAMINT hardware, amplitude scaling must be applied to the convolved signal. As a last step a representation of thermal noise at the receiver following a White Gaussian Noise Distribution is injected into the signal.

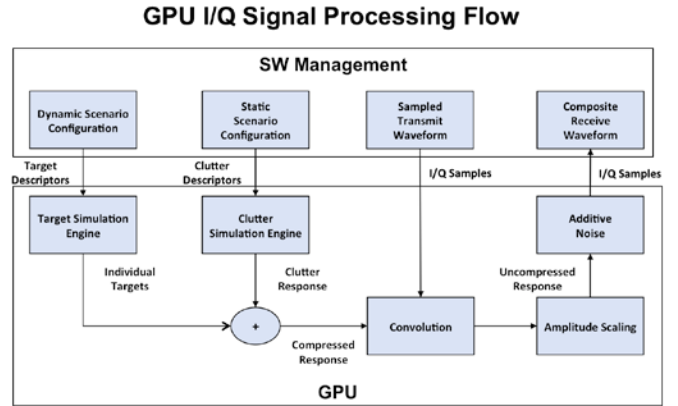


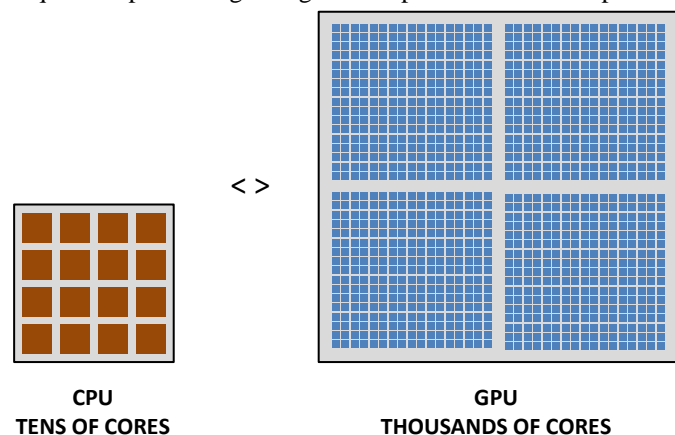
Figure 1: GPU IQ Signal Processing Flow

## III. CUDA-GPU

### GPU Architecture

GPUs were originally designed to efficiently handle graphics rendering applications. Such applications are characterized as being compute intensive and highly parallel.

Based on these features, GPUs have devoted more transistors for data processing as compared to CPUs, which have devoted more transistors for control logic. This leads to GPUs being characterized as a “many-core” chip, which means that the number of cores it contains is on the order of hundreds or thousands. CPUs on the other hand are “multi-core” chips; characterized by having a lower number of cores, usually on the order of tens, at most (see Figure 2). This design makes the GPUs more suitable to address problems that can be expressed as data or task parallel whereas CPUs are more suitable to address sequential computation. To realize the highest performance attainable for a given application we recommend the use of both, a powerful CPU to handle control and sequential processing along with a powerful GPU to process



highly parallel compute intensive portions of the code.

**Figure 2: Multi-core vs. Many-core.**

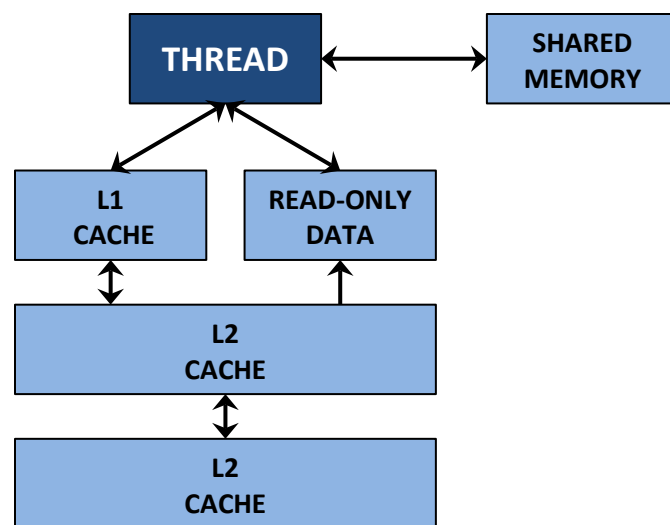
GPUs have a hierarchical architecture in terms of processors, threads, and memory. These are briefly discussed next

#### Processor Hierarchy

The Tesla Kepler architecture contains at most 2880 core processors organized as 15 Streaming Multiprocessors (SMX). Each SMX contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU) for transcendental operations, 64 KB for shared memory and L1 cache, and 48 KB of read only data cache. Each multiprocessor performs computation following the Single Instruction Multiple Thread (SIMT) model, which is similar to the Single Instruction Multiple Data (SIMD) model with the difference being that each SIMT thread is executed independently with its own instruction address and register state. Threads are grouped into thread blocks, which get further assigned to an SMX [2].

#### Thread Hierarchy

Threads, grouped into thread blocks, can be organized in a one, two, or three-dimensional alignment. There are limitations on the maximum number of threads that a block can have, as all threads within a block are expected to reside in the same SMX and must share limited resources. (The CUDA programmer needs to be aware of and take into account such



limitations.) Blocks are further organized in a one or two-dimensional grid. The number of thread blocks within a grid is usually dictated by the size of the data being processed or by hardware limitations of the particular GPU card.

#### Memory Hierarchy

During execution CUDA threads access data via multiple memory spaces. All threads have access to Global Memory, which is off-chip, uncached and has high latency. The lifetime of global memory is the lifetime of the application (unless memory is explicitly de-allocated within the application). The next memory level is Shared Memory. Shared Memory is a low latency on-chip memory, smaller in size compared to global memory, and software managed, meaning that the programmer must explicitly move data in and out of this store. For the Kepler line, the programmer has the option to configure the 64 KB of available memory as a combination of 48 KB/16 KB, 32 KB/32 KB, or 16 KB/48 KB of Shared Memory and L1-Cache, respectively. Kepler also has 48 KB of Read-Only-Cache whose lifetime is the duration of the function. In addition to this, Kepler has an improved L2 cache, key for optimal data sharing across the GPU. Figure 3 shows a summary of the memory hierarchy in the Kepler architecture [2].

**Figure 3: Kepler's Memory Hierarchy [2]**

#### CUDA Programming Model

CUDA is a scalable parallel programming model provided by NVIDIA to exploit the parallel processing power of GPUs for non-graphics applications. CUDA is an extension of the C/C++ language that supports a heterogeneous environment where parallel portions of the application code are sent to the GPU device as kernels and remaining portions of code are run on the CPU host. Kernels are functions that are executed on the GPU by multiple threads. Each kernel call launches a grid of thread blocks specified by the CUDA programmer. In the CUDA execution model, data are transferred from the host to the device, kernel functions get executed on the device, and relevant results are transferred back to the host. Within a kernel, the CUDA programmer has access

to specific blocks and thread via the use of built in CUDA variables [3].

#### IV. CUDA LIBRARY

The T<sup>2</sup>OURNAMINT CUDA Library or T2 CUDA Library, for short, is summarized in Figure 4. It contains two functions to initialize a CUDA capable GPU device and to release its resources at the end of a test. These functions are called once during system startup and once during system shutdown. Once the T2 software is ready to start processing the first of potentially multiple waveforms, it will call the library initialization routine for each waveform to be processed. At this time if clutter and noise modeling are required, respective library engines are also initialized.

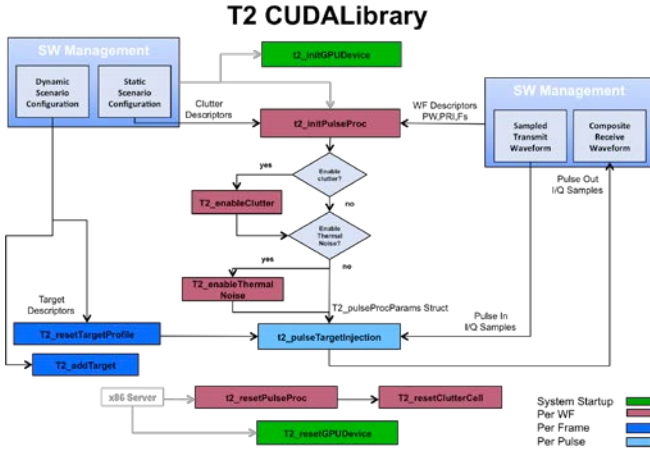


Figure 4: T<sup>2</sup>OURNAMINT CUDA LIBRARY

After initialization of the main library there are two levels of processing that occur, the frame level and the pulse level. The frame level processing latency is limited by the software management system, which runs on a Colfax CX1350s-XXK6 with 2 Intel Xeon E5-2600, 10 cores each. The frame processing speed that is currently achieved is about 10 ms, limiting the frequency at which target positions and gains are being updated throughout the test. In the future more effort will be dedicated toward optimizing the runtime performance of the frame processing; however, for slow moving targets the current frame latency is good enough to execute the tests T<sup>2</sup>OURNAMINT was originally designed for. The T2 CUDA Library includes functions that are called at the frame level to reset previous target range profiles and generate new ones based on the new positions and gains.

On a per-pulse basis we have a different set of limitations and constraints. The GPU is receiving waveform data on a pulse-by-pulse basis at a given rate based on the pulse repetition frequency of the radar. It is very important that we minimize or avoid dropping pulses, as that would lead to inconsistent system under test results. Based on the knowledge of the systems under test we expect to have, we have set a latency goal for the GPU pulse processing library function to fall within 200  $\mu$ s. During this time the waveform pulse will be sent to the GPU, processed on the GPU, and transferred back to the software for further

routing. We are confident that this speed will support the majority of tests that T<sup>2</sup>OURNAMINT must perform.

As the pulse processing routine is key in terms of latency and performance, effort was put into optimizing such functionality. The next section summarizes a set of optimization techniques that were successful in achieving the pulse processing performance goals.

#### V. OPTIMIZATIONS AND RESULTS

##### Baseline

The baseline implementation for T2 CUDA Library focused on designing and implementing the library in a way that exploits precomputation opportunities and that minimizes unnecessary overhead on the pulse-processing routine. This is achieved by having an initialization routine called once per waveform that allocates required GPU memory, initializes and sets up FFT libraries, random number generator engines, and clutter and noise models, as well as other precomputed values that can be reused or easily transformed on a pulse-by-pulse basis. This library initialization routine is accompanied by a de-initialization routine that in turn de-allocates resources allocated during the initialization phase.

##### Pinned Memory

One of the first optimizations to implement in CUDA code is the use of Pinned Memory. Pinned Memory, also known as Non-Pageable memory, is an area of the CPU host DRAM that is page-locked and therefore not swapped with secondary storage. At user level space, allocated memory is always Pageable by default, which provides the user with the notion of unlimited DRAM or a Virtual Memory Space. GPU devices are not able to access Pageable Memory directly, therefore when transferring data to and from the CPU's paged memory space to the GPU's device memory an intermediate data transfer occurs to place the data on the CPU's Non-Pageable memory space, see Figure 5.

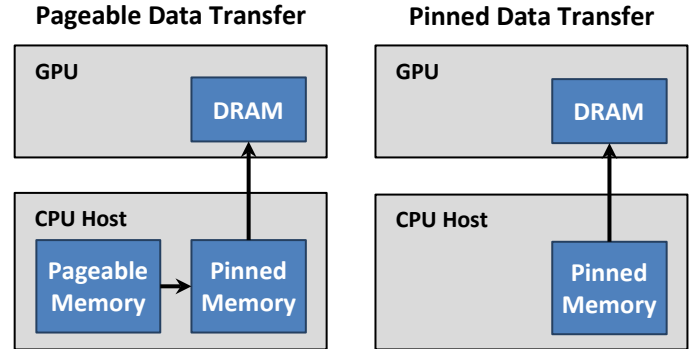


Figure 5: Host and Device Data Transfer

To decrease data transfer time between CPU and GPU, CUDA provides functions to directly allocate CPU host memory in pinned memory space. By using such functions at the host level, we are able to reduce the data transfers to and

from device memory by half, gaining a total speedup of 1.6x on the pulse processing routine.

#### FFT Size Round to Next Power of 2

The pulse processing routines require a series of Fast Fourier Transforms (FFTs) to be executed. In such cases we leverage the use of the cuFFT library included with the NVIDIA CUDA toolkit. The cuFFT library provides a simple interface to leverage the parallel processing power of GPUs for multiple types of FFTs. This library has been highly optimized for data sizes that can be written as a combination of the power of the first 4 prime numbers (2, 3, 5, 7), with smaller prime numbers resulting in better performance [4]. For data sizes that can be written as a power of 2, the cuFFT library will provide the optimal performance available. By rounding data sizes to the next power of 2 we are able to reduce the execution time of the FFTs achieving a total accumulated speedup of 1.9x.

#### Grid Configuration and Padding Optimization

We tried out different grid and block configuration parameters to find an optimal configuration. For our case having 64 threads per thread block gave us the best performance. The optimal number of threads per block that will provide best performance to a kernel is influenced by the amount of resources required for execution such as registers and shared memory. Different applications will have different optimal thread block sizes.

At this stage we also decided to optimize the padding approach for the data to be FFT'ed. In previous versions, if needed, the padding was done before transferring the data to the device, causing the transfer of unnecessary values into the GPU card. To improve this approach, we created a kernel that clears out the memory on the GPU where the data is to be placed, allowing us to only transfer the actual values from the host even if padding is required. In this implementation, if padding is needed it is already taken care of by the kernel that clears the memory before the transfer occurs. The time it takes to clear the values on the device memory before the actual transfer is insignificant, compared to the time it takes to transfer unnecessary values. These optimizations combined, the grid configuration changes along with the padding optimization, led to an overall accumulated speedup of 2.8x.

#### Card Upgrade

Although design decisions for T<sup>2</sup>OURNAMINT included the use of an NVIDIA Tesla K40 card, the best NVIDIA card available at the time for both single and double precision performance, we began development with an NVIDIA Tesla K20 that was readily available. To show the compatibility benefits of using NVIDIA GPU cards as well as the automatic performance gains by switching to a newer card we included the performance results of doing such an upgrade. The card upgrade resulted in a performance gain at the compute level as well as at the bandwidth level. A total of 4.5x speedup was achieved by the upgrade of the card.

#### Stream Support

Newer versions of GPU cards and CUDA have the capability to run multiple kernels concurrently, providing a better way to support task parallelism. The NVIDIA Tesla-Keppler GPU card along with the CUDA Toolkit 7.5, have support for such functionality. This capability to execute multiple kernels at the same time is made possible via the use of CUDA streams. A CUDA stream contains a series of kernel launches or memory transfers that might be dependent on each other or need to be synchronized among them. Functions or kernels running on different streams have seemingly no dependency and can run concurrently. However, if needed, synchronization points between multiple streams can also be achieved via the use of CUDA events. CUDA events act as a lock on the execution of a given stream making the stream wait for a particular event to occur on another stream before resuming execution. The use of CUDA streams not only allows having multiple kernels running concurrently but also aids in hiding communication with computation latency, provided it has not been fully optimized at this stage. With stream support we were able to achieve an overall accumulated speedup of 5.3x for blocking calls and 5.8x for non-blocking pulse processing calls.

## VI. RESULTS

A summary of the optimizations discussed in the previous section, further broken down into time spent in data transfers versus computation time, can be found in Table 1. As mentioned in the introduction, the runtime performance goal for the pulse processing routine was 200  $\mu$ s. By investing time in optimizing this routine we were able to decrease the processing time from about 1ms down to 178  $\mu$ s, which gives us a total speedup of 5.8x. Results show that we were not only able to achieve the runtime performance requirements set out for the pulse processing routine but we were also able to exceed them.

### T2 CUDA Library Optimizations Summary

|                                       | Baseline       | Opt 1         | Opt 2           | Opt 3                      | Opt 4                | Opt 5                  | Opt 5                  |
|---------------------------------------|----------------|---------------|-----------------|----------------------------|----------------------|------------------------|------------------------|
|                                       |                | Pinned Memory | Opt 1 + Power 2 | Opt 2 + Grid Configuration | Opt 3 + Card Upgrade | Opt 4 + Stream Support | Opt 4 + Stream Support |
|                                       |                |               |                 | Padding Optimization       |                      | Blocking               | Non-Blocking           |
|                                       | K20            | K20           | K20             | K20                        | K40                  | K40                    | K40                    |
| Data Movement Time                    | 824 us         | 389 us        | 394 us          | 224 us                     | 112 us               | interleaved            | interleaved            |
| Compute Time                          | 264 us         | 268 us        | 178 us          | 168 us                     | 146 us               | interleaved            | interleaved            |
| Total Speedup                         | n/a            | 1.6x          | 1.9x            | 2.8x                       | 4.5x                 | 5.3x                   | 5.82x                  |
| Total Runtime (No-benchmark overhead) | <u>1037 us</u> | 624 us        | 537 us          | 364 us                     | 231                  | 195 us                 | <u>178 us</u>          |

\*\*Goal = 200us

Table 1: T2 CUDA Library Optimizations Summary

## VII. FUTURE WORK

As future work we recommend the enhancement and further optimization of the T2 CUDA Library by using more powerful cards. The Titan X, which is currently available in the market, although it has a lower double precision performance, ~0.2 TFLOPS as compared to ~1.43 TFLOPS in the NVIDIA

Tesla K40, exhibits higher single precision performance,  $\sim 7$  TFLOPS compared to  $\sim 4$  TFLOPS. To avoid losing double precision performance, we recommend upgrading to the recently announced NVIDIA GTX 1080, the first of the NVIDIA cards with the Tesla-Pascal architecture, Tesla P100 [5]. The card is expected to be available in the summer of 2016. The card will have higher throughput and bandwidth than the latest Tesla K40 and Titan X and features the new NVLINK interconnect that allows faster data transfer rates allowing memory bound applications to also benefit from NVIDIA GPUs accelerators.

#### VIII. SUMMARY

We have successfully implemented and optimized a CUDA library to handle environmental effects in T<sup>2</sup>OURNAMINT for EW testing. We have shown that by dedicating time for optimization, the use of NVIDIA GPUs and CUDA are a feasible tool in accelerating the performance of an application and achieving performance goals in the real-time processing domain. With the optimizations presented in this paper we were not only able to achieve performance requirements but we were able to exceed them. Further performance improvements can be achieved by leveraging the latest of NVIDIA GPU cards and optimizing for such architectures.

#### IX. ACKNOWLEDGEMENTS

We would like to thank our sponsors at NAVAIR Pt. Mugu for supporting this project and allowing us to incorporate the GPGPU technology in the T<sup>2</sup>OURNAMINT system. We also would like to thank the T<sup>2</sup>OURNAMINT team at MIT Lincoln Laboratory for their contributions to this project.

#### REFERENCES

- [1] J. B. Billingsley, "Statistical Analysis of Measured Radar Ground Clutter Data," IEEE Trans. Aerospace and Electronic Systems, vol 35, no. 2, April 1999.
- [2] NVIDIA, "White Paper, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012 NVIDIA Corporation.
- [3] NVIDIA, "CUDA C Programming Guide," 2007-2015 NVIDIA Corporation.
- [4] NVIDIA, "cuFFT User Guide," 2015, NVIDIA Corporation, Retrieved from <http://docs.nvidia.com/cuda/cufft/index.html#axzz49bBKJGXy>.
- [5] NVIDIA, "Whitepaper, NVIDIA TESLA P100," 2016, NVIDIA Corporation.